Redis笔记

Author：Yulin XIE

几个数据类型：

Activation:

```
cd /user/local/bin
redis-server redis-config/redis.conf
redis-cli -p 6379
```

Empty the database:

```
flushall
flushdb
```

String:

```
set key value           #set field:sub-field: value 可以通过这种方式设置出便于json取值的形式
#Exemple: mset user:{id}:{field} 即：mset user:1:name zhangsan user:1:age 18
#或者存为json方式：  set user:1 {name:zhangsan,age:18}
get key
expire key seconds
...
```

Set: 该数据类型很适合放关注列表和粉丝列表，具有唯一性，而且便于求交集，做各种运算

```
sadd key value          #增加元素
smembers key            #列举元素
sismember key value     #判断是否为set中元素
scard key               #获取元素个数
srem key value          #移除元素
sranmember key          #取随机元素
spop key                #取随机元素，并弹出该set
somve key1 key2 value   #将key1中的value移动到key2中
sinter key1 key2        #交集 ∩ （可用于求共同好友/相互关注）
sunion key1 key2        #并集 ∪
sdiff key1 key2         #差集
```

Hash：理解为key-map，即 key-，本质上和String没有很大区别

```
hset key field value                            #设置该key中该field对应的value
hget key field                                  #获取该key中该field对应的value
hmset key field1 value1 field2 value2 ...       #一次性设置该key中多个field-value对
hmget key field1 field2 ...                     #一次性获取该key中多个field-value对
hgetall key                                     #获取该key中所有内容，逐个输出field
value
hdel key field                                  #删除该key中的field，对应的value也删除了
hlen key                                        #求field个数（即hash表字段数量）
hexists key field                               #判断指定字段是否存在
hkeys keys                                      #只获取所有field
hvals keys                                      #只获取所有value
#该数据类型常用来存变更的数据，比如说User类中的Name,Age,Address属性等
#Example：User作为key，各种属性Name,Age,Address作为field，内容作为value
```

zset有序集合
使用场景：
1、其中的排序可以用于比如成绩单排序、工资表排序、或者排行榜应用实现。
2、将score作为权重对消息进行分级，带权重进行判断

```
zadd key score member                           #增加元素
ZCOUNT key -inf +inf                            #计算区间内个数
ZPOPMIN key                                     #弹出最小值
zrange key 0 -1                                 #全部输出
zrem key member                                 #删除key中的member
zcard key                                       #统计个数
ZREVRANGEBYSCORE key +inf -inf withscores       #带上score进行排序并输出
```

Redis的基本事务操作

事务的本质：一组命令的集合。一个事务中的所有命令都会被序列化，在执行过程中会按照顺序执行，且具有排他性。

Redis单条命令保证原子性，但是事务不保证原子性。

Redis事务并没有隔离级别的概念。事务中的所有命令只有在发起执行命令execute的时候才会执行。

Redis的事务：

- 开启事务 multi
- 命令入队
- 执行事务 execute

正常执行事务：

```
127.0.0.1:6379> MULTI           #开启事务
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> get k2
QUEUED
```

```
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> EXEC            #执行事务
1) OK
2) OK
3) "v2"
4) OK
127.0.0.1:6379>  get k2
"v2"
```

放弃事务：

```
127.0.0.1:6379> FLUSHALL        #开启事务
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> get k2
QUEUED
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> DISCARD         #放弃事务
OK
127.0.0.1:6379> get k5
(nil)
```

编译型异常：代码有问题，命令有错，事务中所有命令都不会执行

```
127.0.0.1:6379> FLUSHALL
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> getset k3       #这里是错误的点
(error) ERR wrong number of arguments for 'getset' command
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> EXEC            #执行事务报错
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get k4          #显然无法获取
(nil)
```

运行时异常：事务队列中存在语法性错误，则错误命令抛出异常，其他命令正常执行

```
127.0.0.1:6379> set k1 "v1"                         #字符串k1不支持自增
OK
127.0.0.1:6379> MULTI                               #开启事务
OK
```

```
127.0.0.1:6379> incr k1                                    #自增k1，执行失败
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
#这里也说明了：**Redis单条命令保证原子性，但是事务不保证原子性。**
127.0.0.1:6379> EXEC
1) (error) ERR value is not an integer or out of range  #虽然第一条命令报错了，但是依然正常执行了事务
2) OK                                                   #其余的两条命令执行ok
3) OK
127.0.0.1:6379> get k2                                    #其余的两条命令获取值也ok
"v2"
127.0.0.1:6379> get k3
"v3"
```

监控

有两种不同的制度：

- 乐观锁：认为什么时候都会出问题，无论做什么都会加锁
- 悲观锁：认为什么时候都不会出问题，所以不会上锁。更新数据的时候会去判断一下，在此期间是否有人修改过这个数据。比如在MySQL中会使用version字段：获取version，更新的时候比较version。

比如一个单线程的执行成功的乐观锁：

```
127.0.0.1:6379> FLUSHALL
OK
127.0.0.1:6379> set money 100   #钱是100
OK
127.0.0.1:6379> set out 0       #支出为0
OK
127.0.0.1:6379> WATCH money     #监视money对象
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> DECRBY money 20
QUEUED
127.0.0.1:6379> INCRBY out 20
QUEUED
127.0.0.1:6379> EXEC            #数据在此期间并没有发生变动，这个时候可以正常执行成功
1) (integer) 80                 #钱变成80
2) (integer) 20                 #支出变成20
```

这里演示一个双线程的，第一个线程的事务执行过程中，对象被第二个线程修改，从而导致第一个线程的事务执行失败的案例：

```
127.0.0.1:6379> set money 100
OK
```

```
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379> watch money
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> DECRBY money 20
QUEUED
127.0.0.1:6379> INCRBY out 20
QUEUED
# 在这个时候，第二个线程做了如下操作：
# 127.0.0.1:6379> set money 500
# OK
127.0.0.1:6379> EXEC
(nil)
127.0.0.1:6379> get money
"500"
```

从上面可以看出，我们在Redis中使用`watch`可以作为乐观锁操作。

如果发现事务执行失败，则使用`UNWATCH`进行解锁，然后再`WATCH`获取新的值，再次监视。同样地`EXEC`时对比监视的值是否发生了变化，如果没有变化则可以执行成功，如果又有变化则重复这个过程。 → 这个可以用于秒杀抢购应用。

使用Jedis进行代码开发

建立maven项目，导入包依赖

```
<dependencies>
        <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>3.2.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.68</version>
        </dependency>
</dependencies>
```

打开redis-server并进行本地连接测试

```
public class testPing {
    public static void main(String[] args) {
        Jedis jedis = new Jedis("127.0.0.1",6379);
        System.out.println(jedis.ping());  //返回PONG
    }
}
```

进行事务测试

```
public static void main(String[] args) {
    Jedis jedis = new Jedis("127.0.0.1",6379);
    jedis.flushAll();

    JSONObject jsonObject1 = new JSONObject();
    jsonObject1.put("name","yulin");
    jsonObject1.put("age",18);
    String jsonString1 = jsonObject1.toJSONString();

    JSONObject jsonObject2 = new JSONObject();
    jsonObject2.put("name","fufu");
    jsonObject2.put("age",10);
    String jsonString2 = jsonObject2.toJSONString();

    Transaction transaction = jedis.multi();

    try {
        transaction.set("user1",jsonString1);
        transaction.set("user2",jsonString2);
        int i= 1/0  ;
        transaction.exec();
        System.out.println("success");
    } catch (Exception e) {
        transaction.discard();
        System.out.println("discard");
        e.printStackTrace();
    } finally {
        System.out.println(jedis.get("user1"));
        System.out.println(jedis.get("user2"));
        System.out.println(jedis.get("temp"));
        jedis.close();
    }
}
```

将Jedis与Springboot进行整合

导入依赖:

在pom.xml中有这样的依赖:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

该依赖从属于spring-boot-starter-data系列。里面可以看到有下面的这些具体的东西:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>   <!-- spring-data-redis -->
  <version>2.2.6.RELEASE</version>
  <scope>compile</scope>
  <exclusions>
    <exclusion>
      <artifactId>jcl-over-slf4j</artifactId>
      <groupId>org.slf4j</groupId>
```

```
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>io.lettuce</groupId>
  <artifactId>lettuce-core</artifactId>         <!-- lettuce -->
  <version>5.2.2.RELEASE</version>
  <scope>compile</scope>
</dependency>
```

在springboot2.x之后都是使用了lettuce来代替jedis，里面的netty使得实例可以在多个线程中进行共享，不存在线程不安全的情况，可以减少线程数量，更像NIO模式（Jedis像BIO模式），具体地lettuce里面有这样的依赖：

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-bom</artifactId>
    <version>${netty-version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

我们在`spring-boot-autoconfigure-2.2.6.RELEASE.jar/spring.factories`文件中搜索可以找到和redis相关的自动配置类`RedisAutoConfiguration`：

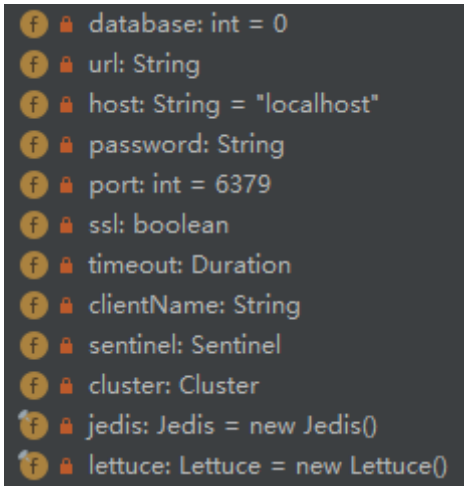
```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class,
JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {
   @Bean
   @ConditionalOnMissingBean(name = "redisTemplate")//这里说明我们可以自定义
redisTemplate来替代这个默认的
   public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory)
         throws UnknownHostException {
       //默认的RedisTemplate没有过多的配置
       //Redis对象都是需要序列化的
       //这里的两个泛型是<Object, Object>，之后的使用需要强转为<String, Object>
      RedisTemplate<Object, Object> template = new RedisTemplate<>();
      template.setConnectionFactory(redisConnectionFactory);
      return template;
   }

   @Bean
   @ConditionalOnMissingBean//由于String是Redis中最常使用的，所以单独搞了一个这个bean
   public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory
redisConnectionFactory)
         throws UnknownHostException {
      StringRedisTemplate template = new StringRedisTemplate();
      template.setConnectionFactory(redisConnectionFactory);
      return template;
   }
```

```
}
```

可以看到与其相匹配的配置文件是`RedisProperties.java`，上方有注释

```
@ConfigurationProperties(prefix = "spring.redis")
```

该类里面的数据成员有：



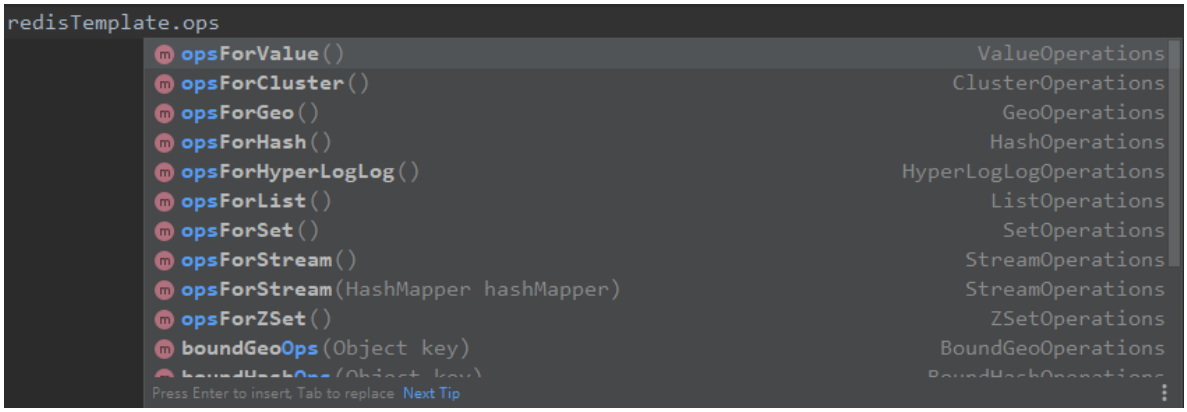
从上面的注释可以知道，我们在`applicationContext.properties`中进行**配置**的时候，前缀是`spring.redis`，比如：

```
spring.redis.host=127.0.0.1
spring.redis.port=6379
```

测试：

```
@SpringBootTest
class RedisSpringbootApplicationTests {
    @Autowired
        RedisTemplate redisTemplate;
    @Test
    void contextLoads() {
        redisTemplate.opsForValue().set("key1", "value1");
        System.out.println(redisTemplate.opsForValue().get("key1"));
    }
}
```

在springboot的测试中我们得到了正确的值，这里的opsForValue用于String类型，还有其他的比如↓，对应于Redis的那六大类型：



除了上面对应各个类型的操作外，一些最基本的操作还可以直接通过RedisTemplate来操作。

通过上面的测试后，我们在RedisTemplate类中还发现了一些序列化工具：

```
private @Nullable RedisSerializer<?> defaultSerializer;
@SuppressWarnings("rawtypes") private @Nullable RedisSerializer keySerializer =
null;
@SuppressWarnings("rawtypes") private @Nullable RedisSerializer valueSerializer
= null;
@SuppressWarnings("rawtypes") private @Nullable RedisSerializer
hashKeySerializer = null;
@SuppressWarnings("rawtypes") private @Nullable RedisSerializer
hashValueSerializer = null;
private RedisSerializer<String> stringSerializer = RedisSerializer.string();
```

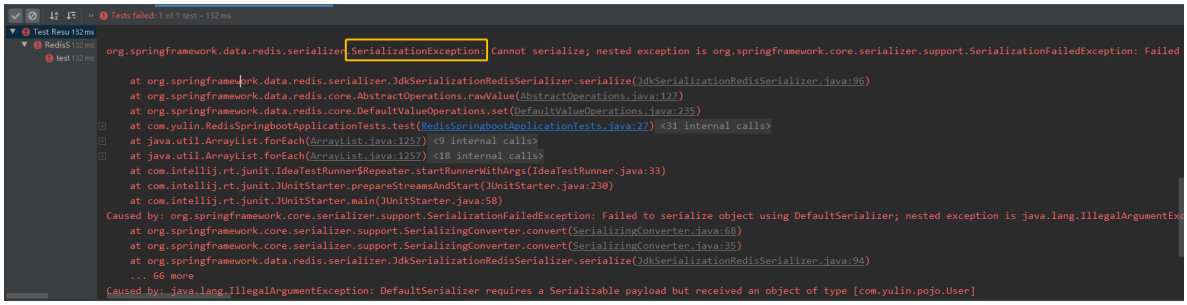
为什么需要这些东西呢？假设我们有一个POJO类User：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Setter
@Getter
@ToString
public class User {
    private String name;
    private int age;
}
```

然后我们进行测试：

```
@Test
void test(){
    User user = new User("name", 18);
    redisTemplate.opsForValue().set("user1",user);
    System.out.println(redisTemplate.opsForValue().get("user1"));
}
```

就会发现报错：序列化错误，无法序列化



（1）而在真实的开发中经常会先序列化成`json`对象，就可以成功：

```
import com.fasterxml.jackson.databind.ObjectMapper;
...
@Test
void test() throws JsonProcessingException {
    User user = new User("name", 18);
    String s = new ObjectMapper().writeValueAsString(user);          //序列化为
json
    redisTemplate.opsForValue().set("user1",s);
    System.out.println(redisTemplate.opsForValue().get("user1"));    // ok
}
```

(2) 另外一种办法就是将POJO的`User`类实现`Serializable`接口，就可以直接放进去Redis:

```
public class User implements Serializable {
    private String name;
    private int age;
}
```

(3) 第三种方法是我们可以**自定义RedisTemplate**：*这是一个通用模板，可以进企业用!*

```
package com.yulin.config;
@Configuration
public class RedisConfig {
    @Bean
    @SuppressWarnings("all")
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory)
            throws UnknownHostException {
        RedisTemplate<String, Object> template = new RedisTemplate<String,
Object>();
        template.setConnectionFactory(redisConnectionFactory);

        //Json序列化配置
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);
        objectMapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(objectMapper);
        //String序列化配置
        StringRedisSerializer stringRedisSerializer = new
StringRedisSerializer();

        //key都使用String的序列化方式
        template.setKeySerializer(stringRedisSerializer);
        template.setHashKeySerializer(stringRedisSerializer);
        //value都使用json的序列化方式
        template.setValueSerializer(jackson2JsonRedisSerializer);
        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        template.afterPropertiesSet();

        return template;
    }
```

```
}
```

那么在这种情况下，我们的User类不需要实现序列化接口也可以顺利地保存到Redis中：

```
@SpringBootTest
class RedisSpringbootApplicationTests {

    @Autowired
    @Qualifier("redisTemplate") //这个注释可以保证获得我们自己定义的RedisTemplate，因为需要重名覆盖底层库
    RedisTemplate redisTemplate;

    /**
     * 这里使用自定义RedisTemplate来进行操作
     */
    @Test
    void test() throws JsonProcessingException {
        User user = new User("yulin", 22);
        redisTemplate.opsForValue().set("yulin",user);
        System.out.println(redisTemplate.opsForValue().get("yulin")); //输出：User(name=yulin, age=22)
    }
}
```

（4）在写了上面的自定义RedisTemplate后，我们实现了序列化的操作，但是在写代码的过程中依然要写诸如`redisTemplate.opsForValue().set()`这样冗长的代码，为了使代码变得简洁，同时做到和命令行一样的API，在企业中一般会开发出自己的Utils工具类，比如这个通用模板：

> 参考自https://blog.csdn.net/weixin_42231507/article/details/80776544，经过改动，未完全测试

```
package com.yulin.utils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.util.CollectionUtils;
import org.springframework.util.StringUtils;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.TimeUnit;

@Component
public class RedisUtils {
    @Autowired
    @Qualifier("redisTemplate") //这里注入了上面写的自定义RedisTemplate！！
    private RedisTemplate<String, Object> redisTemplate;

    /**
     * 指定缓存失效时间
     * @param key 键
     * @param time 时间(秒)
     * @return
```

```
     */
    public boolean expire(String key,long time){
        try {
            if(time>0){
                redisTemplate.expire(key, time, TimeUnit.SECONDS);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 根据key 获取过期时间
     * @param key 键 不能为null
     * @return 时间(秒) 返回0代表为永久有效
     */
    public long getExpire(String key){
        return redisTemplate.getExpire(key,TimeUnit.SECONDS);
    }

    /**
     * 判断key是否存在
     * @param key 键
     * @return true 存在 false不存在
     */
    public boolean hasKey(String key){
        try {
            return redisTemplate.hasKey(key);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 删除缓存
     * @param key 可以传一个值 或多个
     */
    @SuppressWarnings("unchecked")
    public void del(String ... key){
        if (key != null && key.length > 0) {
            if (key.length == 1) {
                redisTemplate.delete(key[0]);
            } else {
                redisTemplate.delete(CollectionUtils.arrayToList(key));
            }
        }
    }

    /**
     * 批量删除<br>
     * （该操作会执行模糊查询，请尽量不要使用，以免影响性能或误删）
     * @param pattern
     */
    public void batchDel(String... pattern){
        for (String kp : pattern) {
```

```
            redisTemplate.delete(redisTemplate.keys(kp + "*"));
        }
    }

    //============================String=============================
    /**
     * 普通缓存获取
     * @param key 键
     * @return 值
     */
    public Object get(String key){
        return key == null ? null : redisTemplate.opsForValue().get(key);
    }

    /**
     * 取得缓存（int型）
     * @param key
     * @return
     */
    public Integer getInt(String key){
        String value = (String) redisTemplate.boundValueOps(key).get();
        if(!StringUtils.isEmpty(value)){
            return Integer.valueOf(value);
        }
        return null;
    }

    /**
     * 取得缓存（字符串类型）
     * @param key
     * @return
     */
    public String getStr(String key){
        return (String) redisTemplate.boundValueOps(key).get();
    }

    /**
     * 取得缓存（字符串类型）
     * @param key
     * @return
     */
    public String getStr(String key, boolean retain){
        String value = (String) redisTemplate.boundValueOps(key).get();
        if(!retain){
            redisTemplate.delete(key);
        }
        return value;
    }

    /**
     * 获取缓存<br>
     * 注：基本数据类型(Character除外)，请直接使用get(String key, Class<T> clazz)取值
     * @param key
     * @return
     */
    public Object getObj(String key){
        return redisTemplate.boundValueOps(key).get();
    }
```

```
/**
 * 获取缓存<br>
 * 注：java 8种基本类型的数据请直接使用get(String key, Class<T> clazz)取值
 * @param key
 * @param retain    是否保留
 * @return
 */
public Object getObj(String key, boolean retain){
    Object obj = redisTemplate.boundValueOps(key).get();
    if(!retain){
        redisTemplate.delete(key);
    }
    return obj;
}

/**
 * 获取缓存<br>
 * 注：该方法暂不支持Character数据类型
 * @param key   key
 * @param clazz 类型
 * @return
 */
@SuppressWarnings("unchecked")
public <T> T get(String key, Class<T> clazz) {
    return (T)redisTemplate.boundValueOps(key).get();
}


/**
 * 普通缓存放入
 * @param key 键
 * @param value 值
 * @return true成功 false失败
 */
public boolean set(String key,Object value) {
    try {
        redisTemplate.opsForValue().set(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将value对象写入缓存
 * @param key
 * @param value
 * @param time 失效时间(秒)
 */
public void push(String key,Object value, long time){
    if(value.getClass().equals(String.class)){
        redisTemplate.opsForValue().set(key, value.toString());
    }else if(value.getClass().equals(Integer.class)){
        redisTemplate.opsForValue().set(key, value.toString());
    }else if(value.getClass().equals(Double.class)){
        redisTemplate.opsForValue().set(key, value.toString());
```

```
        }else if(value.getClass().equals(Float.class)){
            redisTemplate.opsForValue().set(key, value.toString());
        }else if(value.getClass().equals(Short.class)){
            redisTemplate.opsForValue().set(key, value.toString());
        }else if(value.getClass().equals(Long.class)){
            redisTemplate.opsForValue().set(key, value.toString());
        }else if(value.getClass().equals(Boolean.class)){
            redisTemplate.opsForValue().set(key, value.toString());
        }else{
            redisTemplate.opsForValue().set(key, value);
        }
        if(time > 0){
            redisTemplate.expire(key, time, TimeUnit.SECONDS);
        }
    }

    /**
     * 普通缓存放入并设置时间
     * @param key 键
     * @param value 值
     * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
     * @return true成功 false 失败
     */
    public boolean set(String key,Object value,long time){
        try {
            if(time>0){
                redisTemplate.opsForValue().set(key, value, time,
TimeUnit.SECONDS);
            }else{
                set(key, value);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 递增
     * @param key 键
     * @param delta 要增加几(大于0)
     * @return
     */
    public long incr(String key, long delta){
        if(delta<0){
            throw new RuntimeException("递增因子必须大于0");
        }
        return redisTemplate.opsForValue().increment(key, delta);
    }

    /**
     * 递减
     * @param key 键
     * @param delta 要减少几(小于0)
     * @return
     */
    public long decr(String key, long delta){
```

```
        if(delta<0){
            throw new RuntimeException("递减因子必须大于0");
        }
        return redisTemplate.opsForValue().increment(key, -delta);
    }

    //================================Map=================================
    /**
     * HashGet
     * @param key 键 不能为null
     * @param item 项 不能为null
     * @return 值
     */
    public Object hget(String key,String item){
        return redisTemplate.opsForHash().get(key, item);
    }

    /**
     * 获取hashKey对应的所有键值
     * @param key 键
     * @return 对应的多个键值
     */
    public Map<Object,Object> hmget(String key){
        return redisTemplate.opsForHash().entries(key);
    }

    /**
     * HashSet
     * @param key 键
     * @param map 对应多个键值
     * @return true 成功 false 失败
     */
    public boolean hmset(String key, Map<String,Object> map){
        try {
            redisTemplate.opsForHash().putAll(key, map);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * HashSet 并设置时间
     * @param key 键
     * @param map 对应多个键值
     * @param time 时间(秒)
     * @return true成功 false失败
     */
    public boolean hmset(String key, Map<String,Object> map, long time){
        try {
            redisTemplate.opsForHash().putAll(key, map);
            if(time>0){
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
```

```
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key 键
     * @param item 项
     * @param value 值
     * @return true 成功 false失败
     */
    public boolean hset(String key,String item,Object value) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key 键
     * @param item 项
     * @param value 值
     * @param time 时间(秒)  注意:如果已存在的hash表有时间,这里将会替换原有的时间
     * @return true 成功 false失败
     */
    public boolean hset(String key,String item,Object value,long time) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            if(time>0){
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 删除hash表中的值
     * @param key 键 不能为null
     * @param item 项 可以使多个 不能为null
     */
    public void hdel(String key, Object... item){
        redisTemplate.opsForHash().delete(key,item);
    }

    /**
     * 判断hash表中是否有该项的值
     * @param key 键 不能为null
     * @param item 项 不能为null
     * @return true 存在 false不存在
     */
    public boolean hHasKey(String key, String item){
```

```
        return redisTemplate.opsForHash().hasKey(key, item);
    }

    /**
     * hash递增 如果不存在,就会创建一个 并把新增后的值返回
     * @param key 键
     * @param item 项
     * @param by 要增加几(大于0)
     * @return
     */
    public double hincr(String key, String item,double by){
        return redisTemplate.opsForHash().increment(key, item, by);
    }

    /**
     * hash递减
     * @param key 键
     * @param item 项
     * @param by 要减少记(小于0)
     * @return
     */
    public double hdecr(String key, String item,double by){
        return redisTemplate.opsForHash().increment(key, item,-by);
    }

    //============================set============================
    /**
     * 根据key获取Set中的所有值
     * @param key 键
     * @return
     */
    public Set<Object> sGet(String key){
        try {
            return redisTemplate.opsForSet().members(key);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 根据value从一个set中查询,是否存在
     * @param key 键
     * @param value 值
     * @return true 存在 false不存在
     */
    public boolean sHasKey(String key,Object value){
        try {
            return redisTemplate.opsForSet().isMember(key, value);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 将数据放入set缓存
     * @param key 键
```

```
     * @param values 值 可以是多个
     * @return 成功个数
     */
    public long sSet(String key, Object...values) {
        try {
            return redisTemplate.opsForSet().add(key, values);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 将set数据放入缓存
     * @param key 键
     * @param time 时间(秒)
     * @param values 值 可以是多个
     * @return 成功个数
     */
    public long sSetAndTime(String key,long time,Object...values) {
        try {
            Long count = redisTemplate.opsForSet().add(key, values);
            if(time>0) expire(key, time);
            return count;
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 获取set缓存的长度
     * @param key 键
     * @return
     */
    public long sGetSetSize(String key){
        try {
            return redisTemplate.opsForSet().size(key);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 移除值为value的
     * @param key 键
     * @param values 值 可以是多个
     * @return 移除的个数
     */
    public long setRemove(String key, Object ...values) {
        try {
            Long count = redisTemplate.opsForSet().remove(key, values);
            return count;
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
```

```
    }
    //=============================list=================================

    /**
     * 获取list缓存的内容
     * @param key 键
     * @param start 开始
     * @param end 结束  0 到 -1代表所有值
     * @return
     */
    public List<Object> lGet(String key, long start, long end){
        try {
            return redisTemplate.opsForList().range(key, start, end);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 获取list缓存的长度
     * @param key 键
     * @return
     */
    public long lGetListSize(String key){
        try {
            return redisTemplate.opsForList().size(key);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 通过索引 获取list中的值
     * @param key 键
     * @param index 索引  index>=0时， 0 表头，1 第二个元素，依次类推；index<0时，-1，表尾，-2倒数第二个元素，依次类推
     * @return
     */
    public Object lGetIndex(String key,long index){
        try {
            return redisTemplate.opsForList().index(key, index);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 将list放入缓存
     * @param key 键
     * @param value 值
     * @return
     */
    public boolean lSet(String key, Object value) {
        try {
            redisTemplate.opsForList().rightPush(key, value);
```

```
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 将list放入缓存
     * @param key 键
     * @param value 值
     * @param time 时间(秒)
     * @return
     */
    public boolean lSet(String key, Object value, long time) {
        try {
            redisTemplate.opsForList().rightPush(key, value);
            if (time > 0) expire(key, time);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 将list放入缓存
     * @param key 键
     * @param value 值
     * @return
     */
    public boolean lSet(String key, List<Object> value) {
        try {
            redisTemplate.opsForList().rightPushAll(key, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 将list放入缓存
     * @param key 键
     * @param value 值
     * @param time 时间(秒)
     * @return
     */
    public boolean lSet(String key, List<Object> value, long time) {
        try {
            redisTemplate.opsForList().rightPushAll(key, value);
            if (time > 0) expire(key, time);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
```

```
    /**
     * 根据索引修改list中的某条数据
     * @param key 键
     * @param index 索引
     * @param value 值
     * @return
     */
    public boolean lUpdateIndex(String key, long index,Object value) {
        try {
            redisTemplate.opsForList().set(key, index, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 移除N个值为value
     * @param key 键
     * @param count 移除多少个
     * @param value 值
     * @return 移除的个数
     */
    public long lRemove(String key,long count,Object value) {
        try {
            Long remove = redisTemplate.opsForList().remove(key, count, value);
            return remove;
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }
}
```

有了上面这一坨工具类的支持后，我们进行Redis的操作就有一点变化了：

```
@Test
void testUtils(){
    redisUtils.set("redisutils",new User("redisutils",23));
    System.out.println(redisUtils.get("redisutils"));       //输出
User(name=redisutils, age=23)
}
```

同样地，上面的User同样不需要实现序列化接口。

Redis.conf配置文件详解

1. **对大小写不敏感**

```
# units are case insensitive so 1GB 1Gb 1gB are all the same.
```

2. **可以包含多个配置文件：**

```
################################## INCLUDES
###################################

# Include one or more other config files here.  This is useful if you
# have a standard template that goes to all Redis servers but also need
# to customize a few per-server settings.  Include files can include
# other files, so use this wisely.
#
# Notice option "include" won't be rewritten by command "CONFIG REWRITE"
# from admin or Redis Sentinel. Since Redis always uses the last processed
# line as value of a configuration directive, you'd better put includes
# at the beginning of this file to avoid overwriting config change at
runtime.
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# include /path/to/local.conf
# include /path/to/other.conf
```

3. 网络配置

```
# 绑定的IP
# It is possible to listen to just one or multiple selected interfaces using
# the "bind" configuration directive, followed by one or more IP addresses.
bind 127.0.0.1

# 端口号
# Accept connections on the specified port, default is 6379 (IANA #815344).
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379
```

4. 通用设置

```
# 后台守护进程开启
# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when
daemonized.
daemonize yes

# 后台守护进程开启后需要一个pid文件
# When the server is daemonized, the pid file
# is used even if not specified, defaulting to "/var/run/redis.pid".
pidfile /var/run/redis_6379.pid

# 日志等级设置
# Specify the server verbosity level.
# This can be one of:
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably)
# warning (only very important / critical messages are logged)
loglevel notice

# 日志存放位置及名称
```

```
# Specify the log file name. Also the empty string can be used to force
# Redis to log on the standard output. Note that if you use standard
# output for logging but daemonize, logs will be sent to /dev/null
logfile ""

# 数据库个数
# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16
```

5. RDB持久化设置

```
################################ SNAPSHOTTING
################################
#
# Save the DB on disk:
#
#   save <seconds> <changes>
#
#   Will save the DB if both the given number of seconds and the given
#   number of write operations against the DB occurred.
#
#   In the example below the behaviour will be to save:
#   after 900 sec (15 min) if at least 1 key changed
#   after 300 sec (5 min) if at least 10 keys changed
#   after 60 sec if at least 10000 keys changed
#
#   Note: you can disable saving completely by commenting out all "save"
lines.
#
#   It is also possible to remove all the previously configured save
#   points by adding a save directive with a single empty string argument
#   like in the following example:
#
#   save ""
save 900 1
save 300 10     # 300s内有至少10个key修改则进行持久化
save 60 10000

# 持久化出错仍继续运行
# However if you have setup your proper monitoring of the Redis server
# and persistence, you may want to disable this feature so that Redis will
# continue to work as usual even if there are problems with disk,
# permissions, and so forth.
stop-writes-on-bgsave-error yes
```

6. 对Redis设置密码并使用密码进行登录

```
config set requirepass MyPassWord #设置密码
auth MyPassWord                   #使用密码进行登录
```

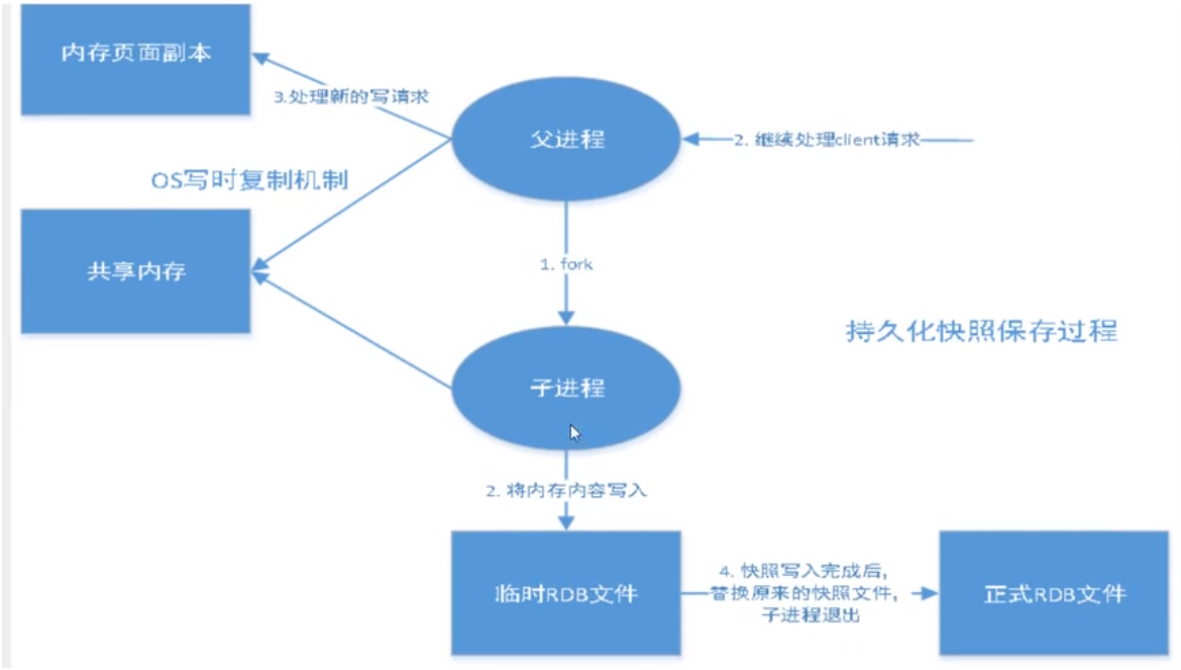
7. APPEND ONLY模式

```
############################## APPEND ONLY MODE
################################

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in
a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.
appendonly no # 默认不开启AOF模式，而是会使用RDB模式进行持久化，大部分情况下够用

# The name of the append only file (default: "appendonly.aof")
appendfilename "appendonly.aof" # 持久化文件名称
```

Redis持久化

RDB RedisDatabase

在指定的时间间隔内将内存中的数据写入硬盘中，恢复时从硬盘中读取快照到内存。



Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的。这就确保了极高的性能。如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。我们默认的就是RDB，一般情况下不需要修改这个配置！

触发产生`dump.rdb`文件的机制：

- save规则满足
- 执行flushall命令
- 退出redis（在命令行中使用shutdown）

特点：

- 适合大规模数据的恢复
- 对数据的完整性要求不高（间歇性备份，中间可能会产生宕机）
- 需要一定的时间间隔进行进程操作，最后一次修改数据可能因为宕机而丢失
- fork进行的时候回占用一定的内存空间

AOF

> 是一种*将**写命令**进行记录*的持久化技术，将每一个收到的写命令都通过write函数追加到文件中。通俗的理解就是**日志记录**。

需要先开启服务：

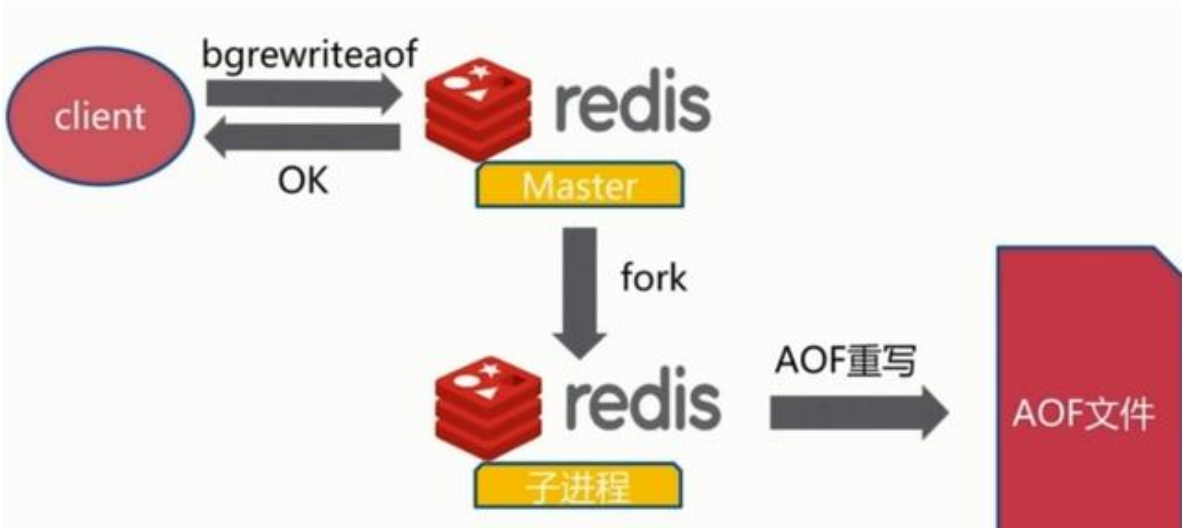
```
appendonly yes
```

会生成一个`appendonly.aof`文件，如果文件被破坏则可以使用`redis-check-aof -fixe`工具进行修复，可以发现在修复后，出错的那条数据也会被删除。

由于这个机制是对数据的无限追加的，所以到达一定程度时会有文件大小的问题，所以通过下面这个选项的设置可以规定到达一定的文件大小后，新建一个aof文件：

```
# Automatic rewrite of the append only file.
# Redis is able to automatically rewrite the log file implicitly calling
# BGREWRITEAOF when the AOF log size grows by the specified percentage.
#
# This is how it works: Redis remembers the size of the AOF file after the
# latest rewrite (if no rewrite has happened since the restart, the size of
# the AOF at startup is used).
#
# This base size is compared to the current size. If the current size is
# bigger than the specified percentage, the rewrite is triggered. Also
# you need to specify a minimal size for the AOF file to be rewritten, this
# is useful to avoid rewriting the AOF file even if the percentage increase
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic AOF
# rewrite feature.

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb  #文件大小为64mb时搞一个新的
```

重写机制如图：

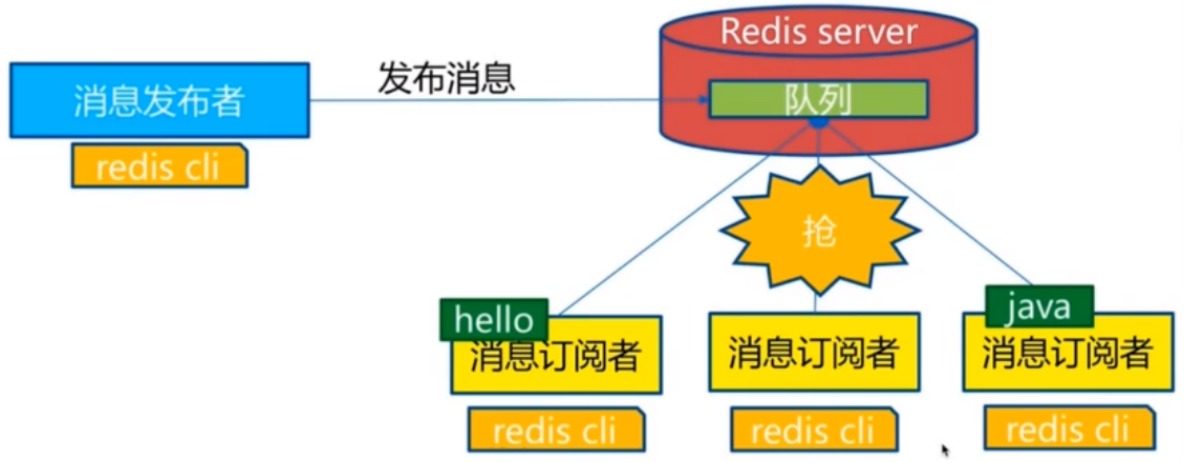


特点：

- AOF可以更好的保护数据不丢失：一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据。
- 对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

Redis发布订阅

这里的发布订阅PUB/SUB是一种消息通信模式。比如微信公众号，微博，关注系统。



订阅端：

```
127.0.0.1:6379> SUBSCRIBE channel #订阅一个频道
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel"
3) (integer) 1

1) "message"    #消息
2) "channel"    #频道
3) "message1"   #消息具体内容
```

发送端：

```
127.0.0.1:6379> PUBLISH channel message1
(integer) 1
```

应用场景：

- 实时消息系统（比如网站的右上角消息提醒）
- 实时聊天（频道作为聊天室，发出的信息回显给所有人即可
- 订阅，关注系统

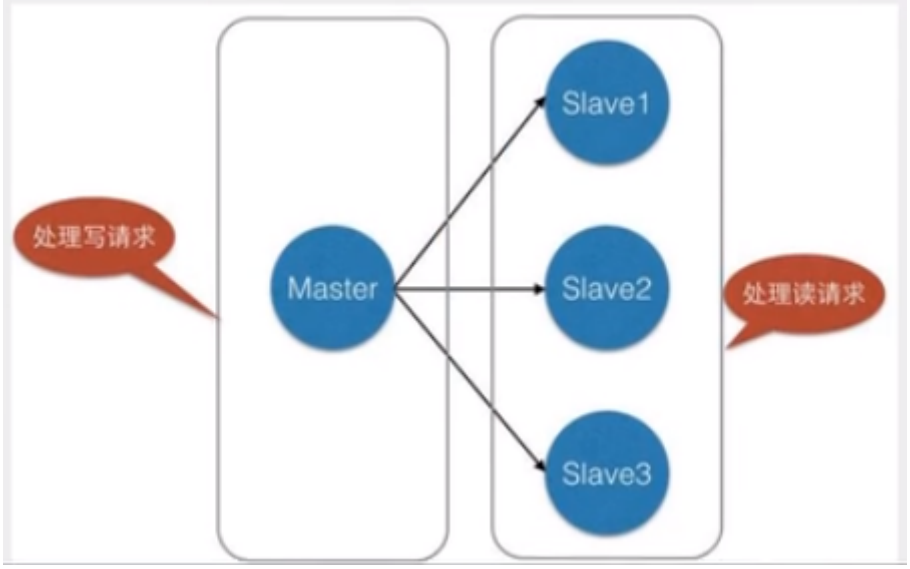
Redis集群环境搭建

Redis主从复制：

将一台Redis服务器的数据复制到其他的Redis服务器。

注意 数据的复制是**单向**的，只能从主节点到从节点。**Master以写为主，Slave以读为主**。这样就做到了**主从复制+读写分离**，这样能够减轻服务器的压力。配置时一般来说最少是一主二从。

- [x] **默认情况下每台Redis服务器都是主节点！** 一般情况下只需要配置从机即可。



主从复制作用：

1. 数据冗余：实现了数据的热备份，是持久化之外的一种数据冗余方式
2. 故障恢复：主节点出问题可以由从节点提供服务，实现快速故障恢复。是一种服务的冗余。
3. 负载均衡：主从复制+读写分离，在读多写少的场景下通过多个从节点分担读负载，可以提高Redis服务器的并发量。
4. 高可用基石：主从复制是哨兵模式和集群能够实施的基础。

环境配置：

只配置从库，不用配置主库。

这里可以先查看当前库的信息：

```
127.0.0.1:6379> info replication #查看当前库的信息
# Replication
role:master          #角色
connected_slaves:0   #没有从机
master_replid:0c7a65764c76b82435cbdc569808ba180927bc03
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

复制三个配置文件，然后修改对应的信息：

1. 端口
2. pid名字
3. log文件名字
4. dump.rdb名字

然后在三个SSH窗口分别开启三个Redis服务，可以得到这样的效果：

```
[root@iZbp1c8miiew8bg4uxq3ndZ bin]# ps -ef | grep redis
root     28442     1  0 22:58 ?        00:00:00 redis-server 127.0.0.1:6379
root     28449     1  0 22:59 ?        00:00:00 redis-server 127.0.0.1:6380
root     28456     1  0 22:59 ?        00:00:00 redis-server 127.0.0.1:6381
root     28469 28404  0 23:00 pts/3    00:00:00 grep --color=auto redis
```

分别开启客户端`redis-cli -p 6379/6380/6381`后，我们在这里将6379作为主机，6380和6381作为从机。因此在从机的窗口处配置：

```
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379
OK
```

然后同样的方法配置6381之后，可以在这两个窗口查看当前库的信息：

```
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379          #这里可以看到配置好的主机
master_link_status:up
master_last_io_seconds_ago:2
master_sync_in_progress:0
slave_repl_offset:14
slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:c80d62073bb3303a3b9f5b9c34af08715877135b
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:14
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:14
```

然后我们在6379的窗口也可以查看到库的信息：

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2      #这里可以看到有两个从机
slave0:ip=127.0.0.1,port=6380,state=online,offset=56,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=56,lag=1
master_replid:c80d62073bb3303a3b9f5b9c34af08715877135b
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:70
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:70
```

这里我们是通过命令行来进行配置，如果想要保持这样的配置，可以直接在.conf文件里面进行修改，好像是这两个：

```
# replica-announce-ip 5.5.5.5
# replica-announce-port 1234
```

主从复制测试：

首先我们在主机查看所有键值对情况，并写入数据：

```
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set k1 v1
OK
```

在主机写入数据前我们在从机中测试所有键值对情况为空，住进写入数据后我们可以获得对应的数据：

```
127.0.0.1:6380> keys *
(empty list or set)
127.0.0.1:6380> keys *
1) "k1"
127.0.0.1:6380> get k1
"v1"
```

与此同时，我们发现并不能在从机处进行写入数据：

```
127.0.0.1:6381> set k2 v2
(error) READONLY You can't write against a read only replica.
```

复制原理：

1. Slave启动成功连接到master后会发送一个sync同步命令
2. master接到命令，启动后台的存盘进程，同时手机所有收到的用于修改数据集的命令，在后台进程执行完毕后，master将传送整个数据文件到slave，并完成一次完全的同步。
3. 全量复制：slave服务在接收到数据库文件数据后，将其存盘并加载到内存中。

4. 增量复制：master继续将新的所有收集到的修改命令依次传给slave，完成同步。
5. 只要重新连接master，一次完全同步（全量复制）将被自动执行。
6. 由上述原理可以得知：
    1. 当我们把主机服务关闭再打开时，主从复制的这个链接依然有效，从机依然可以直接获取到主机写的数据。
    2. 如果是使用命令行进行主从复制配置，那么当从机重启后，其就会变成一台独立的主机。而如果我们再次重新配置好主从复制关系，从机就可以立刻从主机获取到最新的数据。

宕机后手动配置主机：

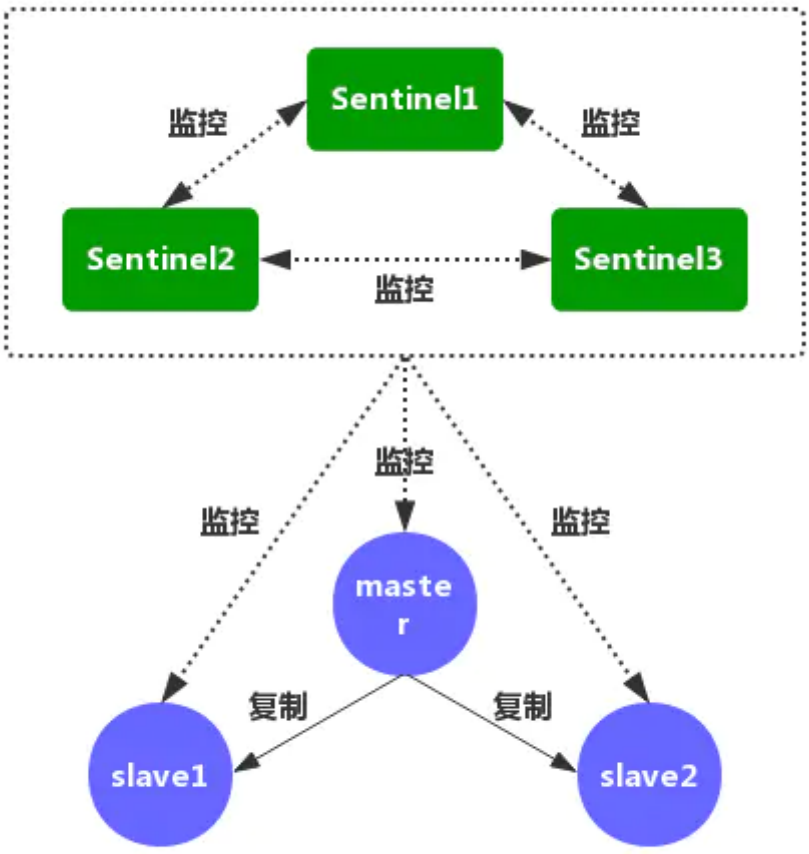
主机挂了，从机可以使用这个命令来变成主机：

```
SLAVEOF NO ONE
```

哨兵模式（重点）：

是一个主从切换技术。以实现主服务器宕机后将从机切换为主服务器的自动化功能。

哨兵是一个独立的进程，原理是**哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。**下图是一个多哨兵模式示例，以避免单个哨兵失效的问题：



Redis Sentinel架构

配置 `sentinel.conf`：

```
#最基础的语法：
#sentinel monitor 主机的名称 host port quorum
sentinel monitor myredis 127.0.0.1 6379 1
#最后面的quorum表示：配置多少个sentinel哨兵统一认为master主节点失联，那么这时客观认为主节点失联了
```

启动哨兵模式服务：

```
[root@iZbp1c8miiew8bg4uxq3ndZ bin]# redis-sentinel redis-config/sentinel.conf #这里要使用上面写的配置文件
28722:X 03 May 2020 02:47:08.100 # oOOOoOOOoOOOo Redis is starting oOOOoOOOoOOOo
28722:X 03 May 2020 02:47:08.101 # Redis version=5.0.8, bits=64, commit=00000000, modified=0, pid=28722, just started
28722:X 03 May 2020 02:47:08.101 # Configuration loaded
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 5.0.8 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in sentinel mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 26379
 |    `-._   `._    /     _.-'    |     PID: 28722
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

28722:X 03 May 2020 02:47:08.102 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
28722:X 03 May 2020 02:47:08.104 # Sentinel ID is 06fc6760774c30f66db885e119810cfe575bbb75
28722:X 03 May 2020 02:47:08.104 # +monitor master myredis 127.0.0.1 6379 quorum 1
28722:X 03 May 2020 02:47:08.105 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:47:08.107 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
```

如果master节点断开了，这时会根据投票算法来在从机中选出一个作为主机：

```
28722:X 03 May 2020 02:49:52.496 # +sdown master myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.496 # +odown master myredis 127.0.0.1 6379 #quorum 1/1
28722:X 03 May 2020 02:49:52.496 # +new-epoch 1
28722:X 03 May 2020 02:49:52.496 # +try-failover master myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.500 # +vote-for-leader 06fc6760774c30f66db885e119810cfe575bbb75 1
28722:X 03 May 2020 02:49:52.500 # +elected-leader master myredis 127.0.0.1 6379
```

```
28722:X 03 May 2020 02:49:52.500 # +failover-state-select-slave master myredis
127.0.0.1 6379
28722:X 03 May 2020 02:49:52.553 # +selected-slave slave 127.0.0.1:6381
127.0.0.1 6381 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.553 * +failover-state-send-slaveof-noone slave
127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.643 * +failover-state-wait-promotion slave
127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.648 # +promoted-slave slave 127.0.0.1:6381
127.0.0.1 6381 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:52.648 # +failover-state-reconf-slaves master myredis
127.0.0.1 6379
28722:X 03 May 2020 02:49:52.714 * +slave-reconf-sent slave 127.0.0.1:6380
127.0.0.1 6380 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:53.747 * +slave-reconf-inprog slave 127.0.0.1:6380
127.0.0.1 6380 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:53.747 * +slave-reconf-done slave 127.0.0.1:6380
127.0.0.1 6380 @ myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:53.799 # +failover-end master myredis 127.0.0.1 6379
28722:X 03 May 2020 02:49:53.799 # +switch-master myredis 127.0.0.1 6379
127.0.0.1 6381
28722:X 03 May 2020 02:49:53.799 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @
myredis 127.0.0.1 6381
28722:X 03 May 2020 02:49:53.799 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @
myredis 127.0.0.1 6381
28722:X 03 May 2020 02:50:23.811 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @
myredis 127.0.0.1 6381
#从最后一行可以看出来选出了新的主机，这次是6381
```

这时如果主机回来了，就只能归并到新的主机下，当做从机，这就是哨兵模式的规则：

```
127.0.0.1:6379> SHUTDOWN     #这就是上面的“master节点断开”
not connected> exit          #退出客户端
#哨兵模式生效后重新开启6379服务器：
[root@iZbp1c8miiew8bg4uxq3ndZ bin]# redis-server redis-config/redis79.conf
[root@iZbp1c8miiew8bg4uxq3ndZ bin]# redis-cli  -p 6379
127.0.0.1:6379> info replication     #查看库信息
# Replication
role:slave                           #此时已经变成了6381的slave
master_host:127.0.0.1
master_port:6381                     #这位6381会兄弟一直保持大哥的地位
...
```

优点：

- 哨兵集群，基于主从复制模式，所有的主从配置优点都具有
- 主从可以切换，故障可以转移，系统的可用性就会更好
- 哨兵模式就是主从模式的升级，手动到自动，更加robust

缺点：

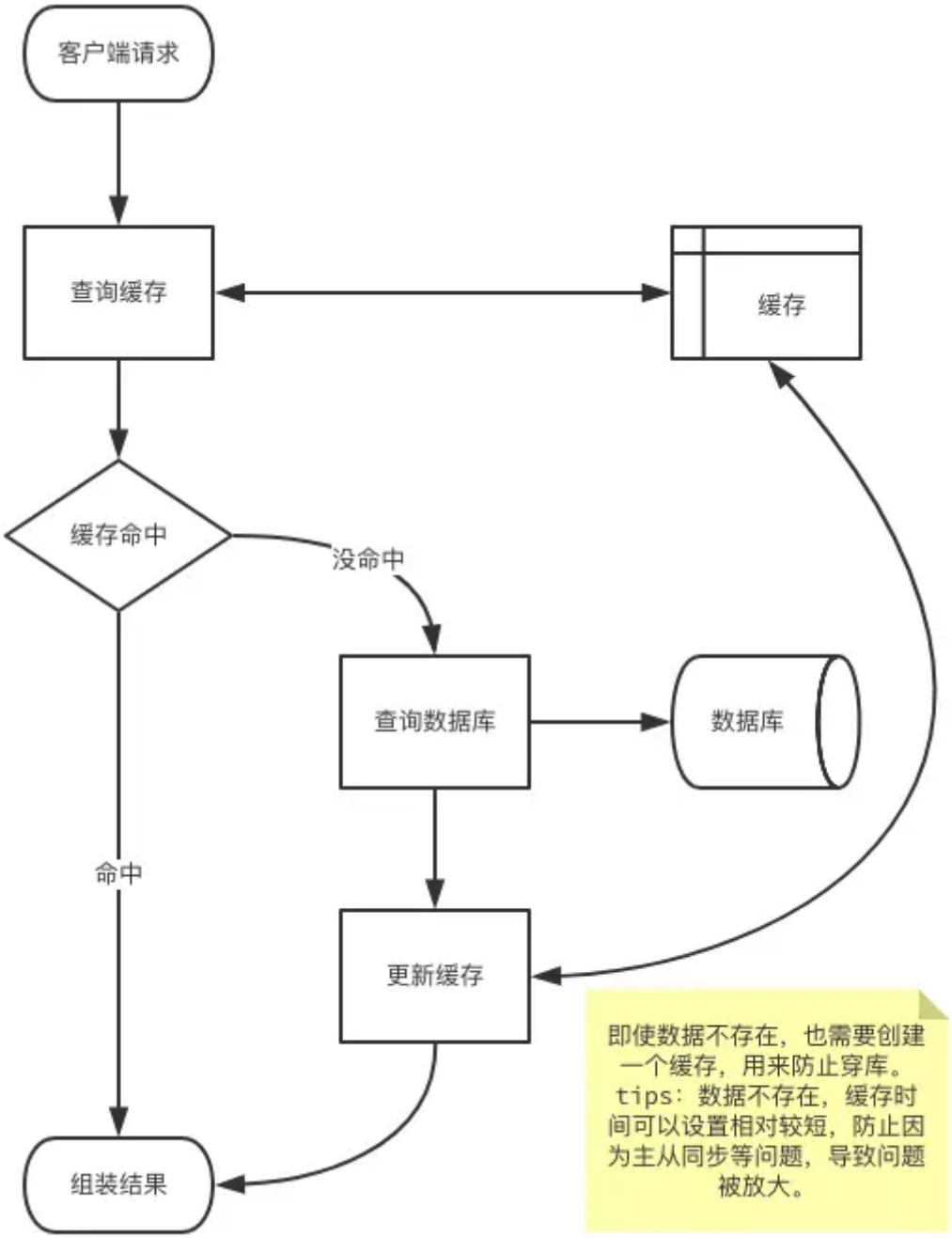
- Redis不容易在线扩容，集群容量一旦到达上限，在线扩容很麻烦
- 实现哨兵模式的配置比较麻烦，里面可选项太多

补充配置选项：

```
#哨兵sentinel实例运行的端口，默认是26379，如果是多哨兵集群模式还要多个配置
port 26379
```

Redis缓存穿透与雪崩 -- 服务的高可用问题

一般的业务缓存过程：



缓存穿透（查不到）：

用户想要查询一个数据，发现Redis内存数据库没有，即缓存没有命中，于是向持久层数据库mysql查询。发现也没有，本次查询失败。当用户很多的时候，缓存都没有命中（比如秒杀），于是都去了请求持久层数据库，这会给持久层数据库造成很大的压力，即出现了缓存穿透。

解决办法：

1. 布隆过滤器：对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合就会丢弃，从而避免了对底层存储系统的查询压力。
2. 缓存空对象：当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据就会从缓存中获取，保护后端数据源。（导致的问题是：1.空值造成浪费。2.过期时间会导致业务的一致性问题）

缓存击穿（查询量太大，缓存过期）：

指的是一个key非常热点，在不停地扛着大并发，大并发集中对这该热点数据进行访问，当这个key在失效的瞬间（还没来得及重新获取），持续的大并发就会突破缓存直接请求数据库，导致数据库瞬间压力过大。

解决办法：

1. 设置热点数据永不过期
2. 加互斥锁。使用分布式锁保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。

缓存雪崩

在某一个时间段，缓存集中过期失效。或者缓存服务器某个节点宕机或者断网。自然形成的缓存雪崩，一定是在某个时间段集中创建缓存。

解决办法：

1. Redis高可用：多增设几台Redis，其实就是搭建的集群，异地多活。
2. 限流降级：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量，比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
3. 数据预热：在正式部署前，把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中，在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

> 笔记来源：
>
> 哔哩哔哩网课 https://space.bilibili.com/95256449 感谢狂神。